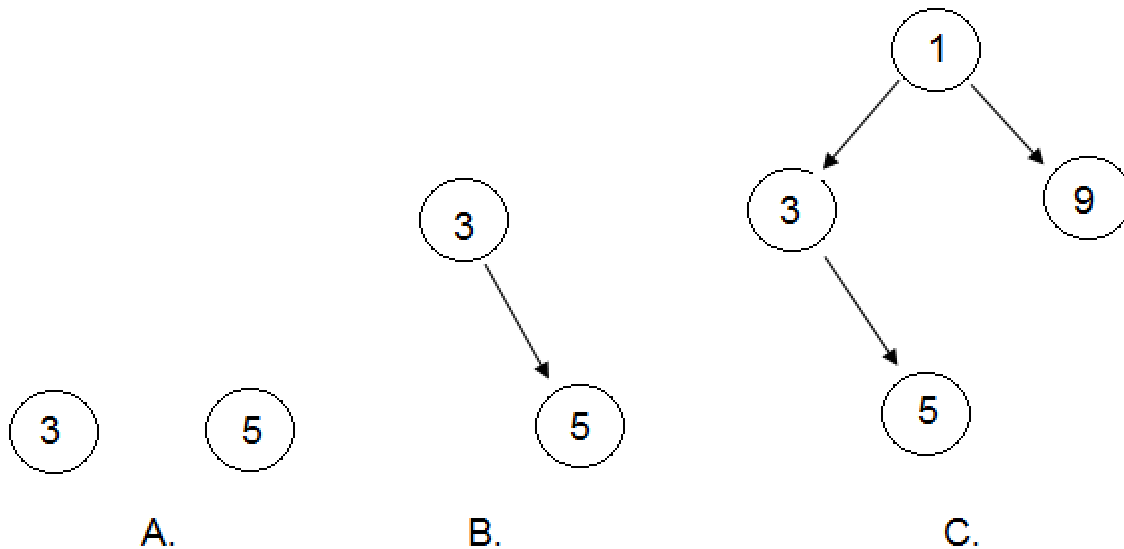
Binomial Heap

The binary heap data structure is fine for the simple operations of inserting, deleting and extracting elements, but other operations aren't so well supported. One such operation is the Union operation, which joins two heaps together. If the heaps are binary heaps then this requires building up a new heap from scratch, using the elements of the old heaps, which is expensive for large heaps. This chapter presents the data structure known as a binomial heap, which supports Union operations more efficiently. Again, binomial heaps can be minimum heaps or maximum heaps, and in this case, the focus is only on minimum heaps.

1. Binomial Trees

The binomial tree is the building block for the binomial heap. A binomial tree is an ordered tree - that is, a tree where the children of each node are ordered. Binomial trees are defined recursively, building up from single nodes. A single tree of degree k is constructed from two trees of degree k - 1 by making the root of one tree the leftmost child of the root of the other tree. This process is shown in Figure below.



Above figure shows the production of a binomial tree. A shows two binomial trees of degree 0. B Shows the two trees combined into a degree-1 tree. C shows two degree-1 trees combined into a degree-2 tree.

2. Binomial Heaps

A binomial heap H consists of a set of binomial trees. Such a set is a binomial heap if it satisfies the following properties:

1. For each binomial tree T in H, the key of every node in T is greater than or equal to the key of its parent.
2. For any integer k ≥ 0, there is no more than one tree in H whose root has degree k.

The algorithms presented later work on a particular representation of a binomial heap. Within the heap, each node stores a pointer to its leftmost child (if any) and its rightmost sibling (if any). The heap itself is a linked list of the roots of its constituent trees, sorted by ascending number of children. The following data is maintained for each non-root node x:

1. key[x] - the criterion by which nodes are ordered,
2. parent[x] - a pointer to the node's parent, or NIL if the node is a root node.
3. child[x] - a pointer to the node's leftmost child, or NIL if the node is childless,
4. sibling[x] - a pointer to the sibling immediately to the right of the node, or NIL if the node has no siblings,
5. degree[x] - the number of children of x.

Binomial Heap Algorithms

(i) Creation

A binomial heap is created with the Make-Binomial-Heap function, shown below. The Allocate-Heap procedure is used to obtain memory for the new heap.

```
Make-Binomial-Heap()
1. H ← Allocate-Heap()
2. head[H] ← NIL
3. return H
```

(ii) Finding the Minimum

To find the minimum of a binomial heap, it is helpful to refer back to the binomial heap properties. Property one implies that the minimum node must be a root node. Thus, all that is needed is a loop over the list of roots.

```
Binomial-Heap-Minimum(Heap)
best_node ← head[Heap]
current ← sibling[best_node]
while current ≠ NIL
do     if key[current] < key[best_node]
        best_node ← current
return best_node
```

(iii) Unifying two heaps

The rest of the binomial heap algorithms are written in terms of heap unification. In this section, the unification algorithm is developed.

Conceptually, the algorithm consists of two parts. The heaps are first joined together into one data structure, and then this structure is is manipulated into satisfying the binomial heap properties.

To address the second phase first, consider two binomial heaps H1 and H2, which are to be merged into H = H1 U H2. Both H1 and H2 obey the binomial heap properties, so in each of them, there is at most one tree whose root has degree k, for k ≥ 0. In H, however, there may be up to two such trees.

To recover the second binomial heap property, such duplicates must be merged. This merging process may result in additional work: when merging a root of degree m with one of degree n, the operation involves adding one as a child of the other - this creates a root with degree p, where p = n + 1 or p = m + 1. However, it is perfectly possible for there to already be a node with degree p, and so another merge is needed. This second merge has the same problem. If root nodes are considered in some arbitrary order, then after every merge, the entire list must be rechecked in case a new conflict has arisen. However, by requiring the list of roots to be in a monotonically increasing order, it is possible to scan through it in a linear fashion. This restriction is enforced by the auxiliary routine Binomial-Heap-Merge:

```
BINOMIAL-HEAP-MERGE(H1,H2){
H ← MAKE-BINOMIAL-HEAP()
if key[head[H2]] < key[head[H1]]
  then head[H] ← head[H2]
     current2 ← sibling[head[H2]]
     current1 ← head[H1]
else head[H] ← head[H1]
     current1 ← sibling[head[H1]]
     current2 ← head[H2]
current ← head[H]
while current1 ≠ NIL and current2 ≠ NIL
     do if key[current1] > key[current2]
       then sibling[current] ← current2
       current ← sibling[current]
       current2 ← sibling[current2]
     else
       sibling[current] ← current1
       current ← sibling[current]
       current1 ← sibling[current1]
if current1 = NIL
     tail ← current2
else tail ← current1
while tail ≠ NIL
do   sibling[current] ← tail
     current ← sibling[current]
     tail ← sibling[tail]
return head[H]
}
```

This routine starts by creating and initializing a new heap, on lines 1 through 9. The code maintains three pointers. The pointer current, stores the root of the tree most recently added to the heap. For the two input heaps, current1 and current2 record their next unprocessed root nodes. In the while loop, these pointers are used to add trees to the new heap, while maintaining the desired monotonic ordering within the resulting list. Finally, the case where the two heaps have differing numbers of trees must be handled - this is done on lines 19 through 25.

Before the whole algorithm is given, one more helper routine is needed. The Binomial-Link routine joins two trees of equal degree:

```
Binomial-Link(Root;Branch)
parent[Branch] = Root
```

```
sibling[Branch] = child[Root]
child[Root] = Branch
degree[Root] = degree[Root] + 1
```

And now the full algorithm :

```
Binomial-Heap-Unify(Heap1,Heap2){
head[Final_Heap] ← Binomial-Heap-Merge(Heap1,Heap2)
if head[Final_Heap] = NIL
    then return Final_Heap
previous ← NIL
current ← head[Final_Heap]
next ← sibling[current]
while next ≠ NIL
    do need_merge ← TRUE
    if (degree[current] ≠ degree[next])
        then need_merge ← FALSE
    if (sibling[next] ≠ NIL and degree[sibling[next]] = degree[next])
        then need_merge ← FALSE
    if (need_merge)
        then if (key[current] ≤ key[next]
            then sibling[current] ← sibling[next]
            Binomial-Link(current; next)
        else if (previous ≠ NIL)
            then sibling[previous] ← next
            Binomial-Link(next,current)
        else head[Final_Heap] ← next
            Binomial-Link(next,current)
    else previous ← current
        current ← next
    next ← sibling[current]
return Final_Heap
}
```

The first line creates a new heap, and populates it with the contents of the old heaps. At this point, all the data are in place, but the heap properties (which are relied on by other heap algorithms) may not hold. The remaining lines restore these properties. The first property applies to individual trees, and so is preserved by the merging operation. As long as Binomial-Link is called with the arguments in the correct order, the first property will never be violated. The second property is restored by repeatedly merging trees whose roots have the same degree.

(iv) Insertion

To insert an element x into a heap H, simply create a new heap containing x and unify it with H:

```
Binomial-Heap-Insert(Heap,Element){
New ← Make-Binomial-Heap()
head[New] ← Element
parent[New] ← Element
sibling[New] ← NIL
child[New] ← NIL
degree[New] ← 1
Binomial-Heap-Unify(Heap,New)
```

```
}
```

(v) Extracting the Minimum

Extracting the smallest element from a binomial heap is fairly simple, due to the recursive manner in which binomial trees are constructed.

```
Binomial-Heap-Extract-Min(Heap){
min ← Binomial-Heap-Minimum(Heap)
rootlist ← Ø
current ← child[min]
while current ≠ NIL
   parent[current] ← NIL
   rootlist ← current + rootlist
new ← Make-Binomial-Heap()
head[new] ← rootlist[0]
Heap ← Binomial-Heap-Unify(Heap;new)
return min
}
```

The only subtlety in the above pseudo-code is on line six, where the next element is added to the front of the list. This is because, within a heap, the list of roots is ordered by increasing degree. (This assumption is behind, for example, the implementation of the Binomial-Heap-Merge algorithm.) However, when a binomial tree is built, the children will be ordered by decreasing degree. Thus, it is necessary to reverse the list of children when said children are promoted to roots.

(vi) Decreasing a key

Decreasing the key of a node in a binomial heap is also simple. The required node has its key adjusted, and is then moved up through the tree until it is no less than its parent, thus ensuring the resulting structure is still a binomial heap.

```
Binomial-Heap-Decrease-Key(Heap, item, key){
key[item] ← key
current ← item
while parent[current] ≠ NIL and key[parent[current]] > key[current]
   tmp ← data[current]
   data[current] ← data[parent[current]]
   data[parent[current]] ← tmp
   current ← parent[current]
}
```

(vii) Deletion

Deletion is simple, given the routines already discussed:

```
Binomial-Heap-Delete(Heap, item){
min ← Binomial-Heap-Minimum(Heap)
Binomial-Heap-Decrease-Key(Heap,item,min - 1)
Binomial-Heap-Extract-Min(Heap)
}
```

Source:

http://www.learnalgorithms.in/#